Em Direção ao Desenvolvimento de Programas Adaptativos Utilizando MDE

[1]S. R. M. Canovas, [2]C. E. Cugnasca

***Abstract* — *Model Driven Engineering* (MDE) é uma abordagem para desenvolvimento de software baseada na transformação de modelos de alto nível, tais como diagramas UML, até a geração automática do código-fonte em uma linguagem alvo. *Set Based Meta Modeling* (SBMM) é um formalismo que permite descrever metamodelos, os quais consistem em um tipo específico de modelos para descrever linguagens de modelagem de software tais como a UML. Este trabalho apresenta um metamodelo para programas adaptativos, definido em SBMM, que estende um metamodelo anterior encontrado na literatura pela adição de restrições. Esta especificação formal de metamodelo é introduzida no software SBMMTool juntamente com uma função de mapeamento para geração de código, que passa então a funcionar como uma ferramenta CASE para edição de modelos de programas adaptativos e abre caminho em direção à aplicação da MDE nesta área.**

***Keywords*— Tecnologias adaptativas (*adaptive technologies*), programas adaptativos (*adaptive programs*), *Model Driven Engineering*, *Set Based Meta Modeling*.**

I. INTRODUÇÃO

Um dispositivo adaptativo é composto por um dispositivo não-adaptativo subjacente e um mecanismo formado por funções adaptativas capaz de alterar o conjunto de regras que define seu comportamento [4]. Esta camada adaptativa confere ao dispositivo capacidade de automodificação, onde as alterações nas regras de comportamento são disparadas em função da configuração corrente do dispositivo e dos estímulos recebidos. Essas alterações se caracterizam pela substituição, inserção ou remoção de regras.

Um autômato adaptativo, por exemplo, é um dispositivo adaptativo que estende o conceito de autômato finito incorporando a característica de desenvolver uma autoreconfiguração em resposta a um estímulo externo [5]. Criação de novos estados e transições em tempo de execução são exemplos de autoreconfiguração.

A tecnologia adaptativa traz uma nova abordagem para resolver problemas, podendo ser aplicada em diversas áreas tais como automação e robótica [5], reconhecedores gramaticais [18] e até mesmo na área agrícola [19].

Um programa adaptativo pode ser entendido como uma especificação de uma sequência automodificável de instruções, que representa um código dinamicamente alterável [3]. Podem ser considerados dispositivos adaptativos onde o dispositivo não-adaptativo subjacente seria um programa estático.

Em um programa adaptativo, as ações adaptativas podem inserir ou remover linhas de código, antes ou depois de processar um estímulo.

Basic Adaptive Language (BADAL) é uma linguagem de programação adaptativa de alto nível proposta por [3]. Uma linguagem adaptativa deve prover instruções explícitas para alteração do código-fonte em tempo de execução, e assim o faz a BADAL. O compilador BADAL apresentado por [3] gera código para o ambiente de execução desenvolvido em [6], que é uma máquina virtual com características específicas que possibilita que um programa realize automodificações em seu código em tempo de execução.

Juntamente com a linguagem BADAL, uma representação gráfica para programas adaptativos é apresentada em [3], descrita em linguagem natural. Cada instância dessa representação gráfica é um modelo, não necessariamente completo, para um programa adaptativo, assim como um diagrama de classes UML é um modelo, também não necessariamente completo, para um sistema orientado a objetos.

Model Driven Engineering (MDE) é uma abordagem para desenvolvimento de software através da qual um sistema é construído pela transformação de modelos em código-fonte em alguma linguagem alvo. Essa transformação pode ocorrer em um número arbitrário de passos. Um modelo de um sistema é uma descrição ou especificação do mesmo considerando um determinado propósito, e por isso pode não representar todos os aspectos e propriedades do sistema por si só [8]. Usualmente um modelo consiste em uma combinação de textos e desenhos, podendo estar descrito em linguagem de modelagem (ex: UML) ou em linguagem natural [1].

Entretanto, para que a MDE seja possível na prática, é necessário que os modelos de software estejam descritos em linguagens que permitam a leitura por programas sendo, portanto, a linguagem natural inadequada para este fim. Os modelos seriam então combinados e transformados em código-fonte em uma linguagem alvo em um processo muito semelhante à compilação de código-fonte em código executável. Com isso, os modelos de software servem não apenas como documentação, mas também como artefatos fundamentais para a construção do programa. Aumenta-se o nível de abstração no desenvolvimento e ganha-se na possibilidade de geração do mesmo sistema para diversas plataformas, uma vez que o mesmo modelo poderia dar origem a código-fonte em mais de uma linguagem alvo, desde que se tenham as transformações necessárias disponíveis.

Embora atualmente a UML seja a linguagem padrão de fato para modelagem de software [9], ela apresenta alguns

[1] S. R. M. Canovas, Escola Politécnica da Universidade de São Paulo, Brasil (e-mail: sergio.canovas@usp.br).
[2] C. E. Cugnasca, Escola Politécnica da Universidade de São Paulo, Brasil (e-mail: carlos.cugnasca@poli.usp.br).

problemas para aplicação da MDE, tais como a dependência de linguagem natural para descrever certos modelos [10] e a falta de semântica clara e precisa em alguns conceitos, que causa ambiguidades em sua interpretação [11]. Além disso, a UML é uma linguagem de modelagem de propósito geral, ou *general purpose language* (GPL), e nem sempre adequada para tipos de aplicação específicos.

Por outro lado, linguagens específicas de domínio, ou *domain specific languages* (DSLs), são linguagens desenhadas para serem úteis em um conjunto específico de tarefas [12]. Elas podem ser combinadas com GPLs para construir conjuntos de modelos de forma mais adequada, ou usadas por si só caso atendam às necessidades.

Seja GPL ou DSL, é fundamental para a MDE a existência de uma forma padronizada de descrever linguagens de modelagem, afinal as transformações de modelos devem ser descritas sobre os conceitos permitidos, e não sobre elementos de modelos específicos. Linguagens de modelagem são descritas por um tipo específico de modelo denominado metamodelo. Existem, por exemplo, metamodelos que descrevem os diagramas da UML. Um metamodelo que descreve uma DSL para modelagem de programas adaptativos é apresentado em [15]. Modelos de *software* devem estar descritos conforme as regras de seus metamodelos pretendidos.

O *Object Management Group* (OMG) provê o *Meta Object Facility* (MOF) como uma linguagem para definir metamodelos, juntamente com a linguagem *Object Constraint Language* (OCL) para estabelecer restrições. Atualmente, existe um consenso geral de que o MOF está subformalizado [13], apresentando alguns problemas similares aos da UML tais como ambiguidade e complexidade. Outros formalismos da literatura [12][13][14] buscam suprir carências, mas normalmente estão focados em aspectos específicos (ex: inferência lógica ou melhorias na definição semântica) e não nos requisitos para aplicação prática da MDE como um todo. Um formalismo alternativo é o *Set Based Meta Modeling* (SBMM) [15], que permite definir metamodelos e modelos precisamente através de equações baseadas em conjuntos, relações e lógica de primeira ordem.

O objetivo deste trabalho é apresentar a utilização de uma ferramenta CASE para o desenvolvimento de programas adaptativos, utilizando o formalismo SBMM para a especificação do metamodelo necessário, dando continuidade ao trabalho apresentado em [15]. As definições do metamodelo são introduzidas na SBMMTool, uma ferramenta meta-CASE. Ferramentas meta-CASE são capazes de gerar ou emular ferramentas CASE customizadas [7], tomando metamodelos como entradas. Com isso, é possível editar modelos de programas adaptativos e ir em direção à aplicação da MDE nesse contexto, na medida em que o modelo pode posteriormente ser transformado em código-fonte BADAL.

A seção II apresenta a linguagem gráfica para representação de programas adaptativos definida por [3]. A seção III descreve um resumo do formalismo SBMM no que diz respeito à definição de metamodelos, enquanto a seção IV apresenta o metamodelo para programas adaptativos introduzido por [15], acrescentando restrições para o mesmo. A seção V apresenta a aplicação deste metamodelo na ferramenta SBMMTool, que passa a se comportar uma ferramenta CASE especializada em programas adaptativos. A seção VI discute os resultados, enquanto a seção VII apresenta as conclusões. Por fim, a seção VIII lista as referências bibliográficas.

II. Programas Adaptativos e Representação Gráfica

Primeiramente será apresentada uma notação para o dispositivo subjacente não-adaptativo do programa adaptativo, isto é, o programa estático escrito em uma linguagem de alto nível hospedeira. A Figura 1 [3] apresenta a arquitetura de um programa projetado como um dispositivo guiado por regras, onde é possível verificar uma camada de código formado por blocos básicos escritos em linguagem hospedeira (camada 1).

A camada 3 provê conexões que ligam a saída de um bloco básico à entrada de algum dos blocos básicos do programa. O valor de saída do bloco recém executado é utilizado por um decisor (camada 2), que encaminha a execução do programa para um dos blocos conectados à saída em função deste valor.

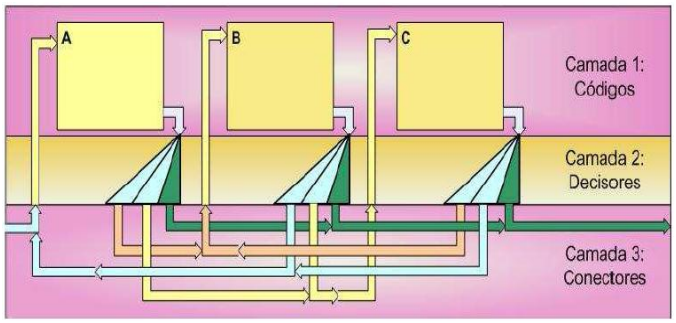

Fig. 1. Arquitetura de um programa não-adaptativo na forma de um dispositivo guiado por regras [3].

Define-se um bloco básico como sendo uma parcela do programa expressa na forma de uma sequência de comandos da linguagem hospedeira, e que deve ser descrito de tal modo que apresente uma só entrada e uma só saída. O valor de saída deve exprimir de alguma forma convencionada uma condição referente ao resultado de sua execução.

Desta forma, os programas adaptativos a serem elaborados pelo programador contêm, como elementos construtivos iniciais, blocos básicos escritos puramente na linguagem hospedeira [3]. Para assegurar a coerência estrutural dos programas assim construídos, é preciso que o programador projete adequadamente as conexões e decisores, e que o compilador faça as validações necessárias.

Cabe ao programador definir os possíveis valores de saída de cada bloco básico. Caso se obtenha um valor de saída não especificado nos decisores, BADAL determina que a cláusula OTHERWISE, obrigatória em todas as conexões, garanta que sempre haverá algum destino para o fluxo do programa após o término da execução de um bloco básico.

Uma entrada de bloco básico pode receber mais de uma conexão, conforme exemplo da Figura 1. Por outro lado, cada valor de saída de um bloco básico deve estar associado a uma única conexão, garantindo que o próximo bloco a executar seja obtido deterministicamente.

Até aqui foi definido o dispositivo não-adaptativo subjacente. A camada adaptativa é introduzida entre a camada de decisores e de conectores. Ela se responsabiliza pela capacidade de alteração do programa em tempo de execução, correspondendo a funções adaptativas. Suas chamadas ficam atreladas às conexões condicionais estabelecidas entre os blocos básicos. A Figura 2 apresenta a arquitetura atualizada com a camada adaptativa.

Na nova camada 3 aparecem todas as funções adaptativas cuja utilização esteja prevista na lógica do programa, e essas são associadas aos conectores da camada 4.

As funções adaptativas devem ser especificadas em algum lugar no corpo do programa adaptativo. A declaração de uma função adaptativa resume-se a indicar as ações de modificação do programa adaptativo, a serem efetuadas em tempo de execução nas ocasiões em que a função for ativada.

BADAL determina que as funções adaptativas se restrinjam a executar ações de inserção ou de remoção, tanto de blocos básicos como de conexões entre eles [3]. As partes do metamodelo que formalizam as funções adaptativas devem refletir esse aspecto.

A referência a um bloco básico deve ser feita por nome, enquanto a referência a uma conexão deve especificar o respectivo bloco básico de origem, bem como o valor de saída desse bloco básico que o seleciona [3].

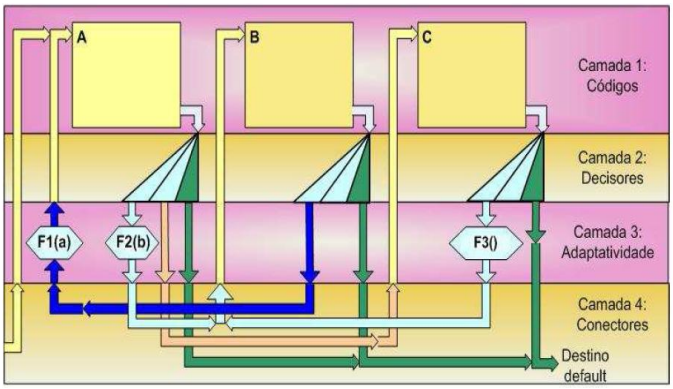

Fig. 2. Arquitetura de um programa adaptativo [3].

A seguinte seção introduzirá o SBMM, e a próxima apresentará um metamodelo formal para a representação gráfica aqui descrita.

III. METAMODELOS EM SBMM

O formalismo SBMM provê meios para descrever metamodelos, modelos e modelos de marcação, operadores de mesclagem de metamodelos, entre outros recursos. Nesta seção será apresentado um resumo do que se refere a metamodelos.

Um metamodelo no SBMM consiste em um nome (cadeia de símbolos) $name_{MM}$, um conjunto C de metaclasses, um conjunto Γ de generalizações, um conjunto E de enumerações e um conjunto R de restrições. Ou seja, um metamodelo é uma quíntupla MM:

$$MM = (name_{MM}, C, \Gamma, E, R) \quad (1)$$

Em que:

- $name_{MM} \in \Sigma^+$, onde Σ é um conjunto de símbolos pré-definido (fora da definição de MM) para a formação de cadeias que representam nomes. Por exemplo, Σ pode ser o conjunto das letras alfabeto romano com maiúsculas e minúsculas, dígitos de 0 a 9 e símbolos adicionais como *underscore*. O elemento $name_{MM}$ representa o nome do metamodelo, servindo para implementar capacidade equivalente a dos identificadores do MOF.
- $C = \{c_1, ..., c_n\}$ é o conjunto finito, eventualmente vazio, de metaclasses. Uma metaclasse representa uma abstração de um conceito de modelagem. Por exemplo, na UML, os conceitos de classe, associação, caso de uso e ator são metaclasses. As classes, associações, casos de uso e atores presentes em um modelo UML de um software são instâncias dessas metaclasses.
- $\Gamma \subset C \times C$ é uma relação não reflexiva livre de ciclos que mapeia submetaclasses em suas supermetaclasses, representando o conceito de herança da orientação a objetos. Não pode ser reflexiva, pois uma metaclasse não é submetaclasse dela mesma. Por não permitir ciclos nem reflexão, então Γ é subconjunto próprio de $C \times C$ desde que $C \neq \varnothing$. A relação Γ indica as relações diretas de herança, enquanto seu fecho transitivo Γ^+ contém também as relações indiretas.
- $E = \{e_1, ..., e_m\}$é o conjunto finito, eventualmente vazio, de enumerações.
- $R = \{r_1,..., r_l\}$ é o conjunto finito, eventualmente vazio, de restrições sobre modelos baseados em MM. Cada restrição é uma sentença em lógica de primeira ordem em que metaclasses e enumerações são vistas como predicados unários e propriedades são enxergadas como enumerações. A expressão lógica $r_1 \wedge r_2 \wedge ... \wedge r_l$ não pode consistir uma falácia, caso contrário qualquer modelo baseado em MM seria não conforme. Um melhor entendimento sobre as restrições será possibilitado através de exemplos mais adiante na seção IV.

A metaclasse a partir da qual um elemento de modelo foi instanciado é denominada de seu tipo base (*base type*). As supermetaclasses dessa metaclasse, e também ela mesma, são denominadas de tipos do elemento (*type*). Logo, o tipo base também um tipo do elemento.

Cada metaclasse c_i possui um nome w_i e um conjunto de propriedades P_i. Sendo assim, podemos defini-las como pares ordenados:

$$c_i = (w_i, P_i) \quad (2)$$

Em que:

- $w_i \in \Sigma^+$ é a cadeia de símbolos que nomeia c_i.
- $P_i = \{p_{i1}, ..., p_{ik}\}$ é o conjunto finito de propriedades, eventualmente vazio.

Os nomes devem ser únicos dentro do metamodelo, tais que se $c_i \neq c_j$ então $w_i \neq w_j$.

Do ponto de vista semântico, as propriedades $p_{ij} \in P_i$ definem *slots* de informação para elementos de modelos (instâncias) da metaclasse C_i e de suas submetaclasses.

Cada propriedade p_{ij} consiste em um nome, multiplicidade, tipo alvo (que pode ser uma metaclasse ou enumeração) e uma multiplicidade que restringe quantos elementos o *slot* de informação representado consegue armazenar. Isto é:

$$p_{ij} = (v_{ij}, t_{ij}, m_{ij}) \quad (3)$$

Em que:

- $v_{ij} \in \Sigma^+$ é a cadeia de símbolos que nomeia p_{ij}.
- $t_{ij} \in C \cup E$ é o tipo alvo da propriedade, ou seja, pode ser uma metaclasse ou uma enumeração do metamodelo.
- $m_{ij} \in \mathbf{N} \times (\mathbf{N}^+ \cup \{*\})$ é a multiplicidade da propriedade, sendo um par ordenado cujo primeiro elemento é um número natural que denota o limite inferior de *slots* que a propriedade é capaz de armazenar. O segundo elemento pode ser um número natural positivo ou um símbolo *, que representa infinito, denotando o limite superior de *slots*. Se $m_{ij} = (1,1)$, por exemplo, isso significa que a propriedade possui um e apenas um *slot* de informação. Se $m_{ij} = (0,*)$, então a propriedade representa uma lista com um número arbitrário de *slots*. Na UML e no MOF as multiplicidades não necessariamente estão dentro de uma faixa de valores inicial e final, podendo ser valores arbitrários tais como 1, 3 e 5 (sem incluir 2 e 4). Mas para os propósitos do SBMM essa definição é suficiente. Para tornar a notação mais semelhante a da UML e MOF, um par ordenado de multiplicidade (x,y) também pode ser denotado por $x..y$.

Os nomes das propriedades devem ser únicos dentro da metaclasse, tais que se $p_{ij} \neq p_{ik}$ então $v_{ij} \neq v_{ik}$.

As enumerações $e_i \in E$ servem para definir tipos de dados básicos cujas instâncias podem assumir um valor de um conjunto finito, definido pela própria enumeração. Ou seja, cada enumeração pode ser definida por:

$$e_i = (u_i, L_i) \quad (4)$$

Em que:

- $u_i \in \Sigma^+$ é a cadeia de símbolos que nomeia e_i.
- L_i é o conjunto finito de valores que as instâncias de e_i podem assumir.

Os nomes devem ser únicos dentro do metamodelo, tais que se $e_i \neq e_j$ então $u_i \neq u_j$.

Por exemplo, pode-se definir uma enumeração de um tipo básico booleano como:

$$e_1 = (\text{“Boolean”}, \{true, false\}) \quad (5)$$

As cadeias de símbolos que representam nomes são denotadas entre aspas para evitar confusão com referências a outros elementos do metamodelo ou valores permitidos em enumerações, ou seja, não confundir a propriedade *p* com o eventual nome (cadeia de símbolos) de um elemento do metamodelo “p”.

Estender a definição de enumeração para aceitar conjuntos L_i infinitos é simples, mas não faz sentido na prática pois uma variável de computador é sempre armazenada em um número finito de bits, e portanto uma instância de enumeração na prática não pode assumir um dentre infinitos valores. Sendo assim, um tipo básico de número inteiro de 64 bits com sinal pode ser definido pela seguinte enumeração:

$$e_2 = (\text{“Integer64”}, \{x \mid (x \in \mathbf{Z}) \wedge (-2^{63} \leq x \leq 2^{63} - 1)\}) \quad (6)$$

Uma enumeração para o tipo String sobre um alfabeto Σ pode ser definida por:

$$e_3 = (\text{“String”}, \{x \mid (x \in \Sigma^*) \wedge (|x| \leq h)\}) \quad (7)$$

A restrição $|x| \leq h$ para algum limitante superior *h* garante que a lista seja finita. Na prática, as linguagens de programação permitem variáveis do tipo String de tamanhos bastante elevados, de forma que *h* é muito grande. Normalmente é limitado pela memória disponível ou alguma característica da linguagem de programação utilizada.

Entretanto, como o SBMM é independente de implementação e usa apenas conceitos teóricos, para facilitar definições poderão ser utilizadas enumerações com infinitos valores permitidos, tais como:

$$e_4 = (\text{“Integer”}, \mathbf{Z}) \quad (8)$$

Isso porque, do ponto de vista teórico, não há impedimentos para que uma enumeração permita infinitos valores possíveis. Até mesmo conjuntos infinitos não enumeráveis poderiam ser usados, como por exemplo:

$$e_5 = (\text{“Real”}, \mathbf{R}) \quad (9)$$

O leitor deve subentender que, ao ser transportado para uma implementação computacional, os eventuais conjuntos infinitos de valores de enumerações deverão ser substituídos por equivalentes práticos. Por exemplo, a enumeração e_2 apresentada acima é uma implementação usual de e_4, já que inteiros de 64 bits resolvem a maioria dos problemas que precisam manipular inteiros em geral.

As enumerações não contêm propriedades e nem possuem uma relação de generalização.

Cada elemento do conjunto *R* de restrições é uma sentença em lógica de primeira ordem que impõe alguma restrição pretendida para os modelos. A seção IV clarificará como funcionam as restrições através de exemplos.

IV. METAMODELO PARA PROGRAMAS ADAPTATIVOS

Para facilitar o entendimento do metamodelo proposto, as sentenças que o compõem serão apresentadas ao longo das explicações. Esse metamodelo foi originalmente introduzido em [15], mas não apresentava o conjunto *R*. Este trabalho acrescenta este conjunto e seus elementos, explicando a interpretação de cada sentença restritiva.

Seja MM_{PA} um metamodelo que descreve a representação de programas adaptativos apresentada na seção II.

$$MM_{PA} = (name_{MM_PA}, C, \Gamma, E, R) \quad (10)$$

Inicialmente define-se $name_{MM_PA}$ = "AdaptiveProgram Metamodel" como o nome do metamodelo. Introduz-se então no conjunto E duas enumerações de tipos básicos e_1 e e_2, já discutidas anteriormente. Os índices foram trocados para manter a sequência dentro deste metamodelo.

- $e_1 = (\text{"String"}, \{x \mid (x \in \Sigma^*) \wedge (|x| \leq h)\})$
- $e_2 = (\text{"Integer"}, \mathbf{Z})$

Essas enumerações serão referenciadas como tipos de propriedades nas metaclasses.

Conforme mostrado na Figura 2, identifica-se que um programa adaptativo é composto por quatro componentes fundamentais: bloco básico, decisor, função adaptativa e conexão. Serão criadas metaclasses para abstrair esses conceitos, bem como metaclasses auxiliares.

Seja $c_1 \in C$ a metaclasse que representa um bloco de código básico. Por convenção, todos os nomes de metaclasse serão iniciados com o prefixo *MC*.

- $c_1 = (\text{"MCBasicBlock"}, P_1)$
  - $P_1 = \{p_{11}, p_{12}\}$
    - $p_{11} = (\text{"Name"}, e_1, 1..1)$
    - $p_{12} = (\text{"Code"}, e_1, 1..1)$

Um bloco básico é composto por um nome e um código-fonte na linguagem hospedeira, neste caso BADAL, ambos representados como as propriedades do tipo String p_{11} e p_{12}. Neste metamodelo apresentado por [15] não se entra no mérito das regras sintáticas do código na linguagem hospedeira, supondo que este seja um problema à parte. O foco será dado aos aspectos arquiteturais do modelo do programa adaptativo apresentados na Figura 2.

O nome do bloco básico não pode ser vazio, e para isso introduz-se a restrição $r_1 \in R$:

- r_1: $\forall o \mid \text{MCBasicBlock}(o) \rightarrow \neg (\text{MCBasicBlock.Name}(o) = \varepsilon)$

A interpretação de r_1 é que para todo elemento do modelo *o*, se *o* for do tipo *MCBasicBlock*, então o valor atribuído à propriedade *Name* de *o* não é vazio. O símbolo ε denota a cadeia (*string*) vazia. Como pôde ser visto, as metaclasses (neste caso *MCBasicBlock*) são enxergadas como predicados unários nas expressões das sentenças de *R*. O mesmo vale para enumerações. As propriedades, referenciadas pelo nome da metaclasse seguido por um ponto e seu próprio nome (neste caso *MCBasicBlock.Name*), são funções que retornam o valor contido no *slot* correspondente no elemento do modelo passado como argumento. Caso a propriedade tenha multiplicidade maior que a unitária, então é retornado um conjunto de valores.

Também não pode haver blocos básicos com o mesmo nome dentro do mesmo modelo. Nesse caso a restrição $r_2 \in R$ exerce esta imposição:

- r_2: $\forall o_1, o_2 \mid \text{MCBasicBlock}(o_1) \wedge \text{MCBasicBlock}(o_2) \wedge (\text{MCBasicBlock.Name}(o_1) = \text{MCBasicBlock.Name}(o_2)) \rightarrow o_1 = o_2$

Interpreta-se r_2 de forma que para todos os elementos do modelo o_1 e o_2, se ambos forem do tipo *MCBasicBlock* e seus nomes forem iguais, então obrigatoriamente $o_1 = o_2$.

Seja $c_2 \in C$ a metaclasse que abstrai a conexão adaptativa, que conecta a saída de um bloco básico a um ou mais blocos básicos conforme valores de saída.

- $c_2 = (\text{"MCAdaptiveConnection"}, P_2)$
  - $P_2 = \{p_{21}, p_{22}, p_{23}\}$
    - $p_{21} = (\text{"From"}, c_1, 1..1)$
    - $p_{22} = (\text{"ConditionalConnections"}, c_3, 0..*)$
    - $p_{23} = (\text{"OtherwiseConnection"}, c_4, 1..1)$

A propriedade p_{21} representa o bloco básico de origem da conexão, enquanto p_{22} referencia as possíveis múltiplas conexões condicionais que partem do bloco. Todo bloco deve ter uma e apenas uma conexão *default* caso a saída em tempo de execução fornecida pelo bloco não corresponda a nenhuma conexão condicional. Essa única conexão é representada por p_{23}.

As propriedades p_{22} e p_{23} definem *slots* de informação para instâncias de c_3 e c_4 respectivamente. Essas, por sua vez, são metaclasses que representam uma conexão condicional e uma conexão *default*, na ordem. Uma conexão condicional nada mais é que uma conexão *default* acrescida de um valor que define que ela será acionada quando a saída do bloco for igual a este valor. Como possuem aspectos comuns, pode-se definir c_4 inicialmente e depois criar c_3 como submetaclasse de c_4.

- $c_4 = (\text{"MCGeneralConnnection"}, P_4)$
  - $P_4 = \{p_{41}, p_{42}, p_{43}\}$
    - $p_{41} = (\text{"To"}, c_1, 1..1)$
    - $p_{42} = (\text{"AdaptiveFuncCallBefore"}, c_5, 0..1)$
    - $p_{43} = (\text{"AdaptiveFuncCallAfter"}, c_5, 0..1)$
- $c_3 = (\text{"MCConditionalConnection"}, P_3)$
  - $P_3 = \{p_{31}\}$
    - $p_{31} = (\text{"OutputValue"}, e_2, 1..1)$

Para que a metaclasse da conexão condicional c_3 seja de fato submetaclasse de c_4, é necessário introduzir o par ordenado (c_3, c_4) em Γ. Até aqui, portanto, $\Gamma = \{(c_3, c_4)\}$. A semântica do SBMM estabelece que c_3 herda as propriedades de c_4.

Observar que a propriedade p_{31}, que representa o valor de saída que decide a utilização da conexão é definido como tipo inteiro (e_2), de acordo com as definições de [3].

Para não haver ambiguidades nos modelos, é necessário estabelecer uma restrição que indique que não pode haver mais de uma conexão condicional baseada no mesmo valor de saída partindo do mesmo bloco básico. Para isso, introduz-se r_3:

- r_3: $\forall a, o_1, o_2 \mid$ MCAdaptiveConnection(a) $\wedge$ MCConditionalConnection(o_1) $\wedge$ MCConditionalConnection(o_2) $\wedge$ ($o_1 \in$ MCAdaptiveConnection.ConditionalConnections(a)) $\wedge$ ($o_2 \in$ MCAdaptiveConnection. ConditionalConnections(a)) $\wedge$ (MCConditionalConnection.OutputValue(o_1) = MCConditionalConnection.OutputValue(o_2)) $\rightarrow o_1 = o_2$

Interpreta-se r_3 de modo que para todos os elementos do modelo o_1 e o_2, se ambos forem do tipo *MCConditionalConnection* e pertencerem ao mesmo objeto a do tipo *MCAdaptiveConnection*, caso seus valores condicionantes sejam iguais então obrigatoriamente $o_1 = o_2$.

Aparece nas definições de p_{42} e p_{43} a metaclasse c_5, ainda não mencionada. Essa metaclasse deve ser definida de modo a representar uma chamada de função adaptativa. Este tipo de chamada se caracteriza por uma função adaptativa alvo e uma lista ordenada, eventualmente vazia, de blocos básicos passados como parâmetros, conforme define [3]. Para refletir esses aspectos, cria-se então a metaclasse $c_5 \in C$.

- c_5 = (*"MCAdaptiveFunctionCall"*, P_5)
  - P_5 = {p_{51}, p_{52}}
    - p_{51} = (*"AdaptiveFunction"*, c_7, 1..1)
    - p_{52} = (*"ParametersValues"*, c_6, 0..*)

Uma vez que a multiplicidade da propriedade p_{52} é 0..*, o que significa que a chamada pode passar um número arbitrário de parâmetros (inclusive nenhum), é necessário introduzir a metaclasse c_6 que representa um par nome/valor, evitando não determinismos caso haja mais de um parâmetro sendo passado. Os nomes dos parâmetros da chamada devem corresponder aos nomes dos parâmetros de entrada definidos na especificação da função adaptativa. Mais adiante, a restrição r_9 garantirá este requisito.

- c_6 = (*"MCAdaptiveFuncParamValue"*, P_6)
  - P_6 = {p_{61}, p_{62}}
    - p_{61} = ("ParameterName", e_1, 1..1)
    - p_{62} = ("ParameterValue", c_1, 1..1)

Na mesma chamada de função adaptativa, não podem ser passados dois parâmetros com o mesmo nome.

- r_4: $\forall a, o_1, o_2 \mid$ MCAdaptiveFunctionCall(a) $\wedge$ MCAdaptiveFuncParamValue(o_1) $\wedge$ MCAdaptiveFuncParamValue(o_2) $\wedge$ ($o_1 \in$ MCAdaptiveFunctionCall.ParametersValues(a)) $\wedge$ ($o_2 \in$ MCAdaptiveFunctionCall. ParametersValues(a)) $\wedge$ (MCAdaptiveFuncParamValue.ParameterName(o_1) = MCAdaptiveFuncParamValue. ParameterName (o_2)) $\rightarrow o_1 = o_2$

A construção da expressão r_4 é exatamente igual à construção de r_3, trocando-se apenas os nomes das metaclasses e propriedades. Interpreta-se da mesma maneira.

Continuando a criação do metamodelo, estabelece-se a metaclasse c_7 para representar as funções adaptativas em si, lembrando que c_5 representa apenas a chamada de uma função adaptativa associada a um conector, e não a especificação da função em si. A metaclasse c_7 fará esse papel.

- c_7 = (*"MCAdaptiveFunction"*, P_7)
  - P_7 = {p_{71}, p_{72}, p_{73}}
    - p_{71} = (*"Name"*, e_1, 1..1)
    - p_{72} = (*"ParametersNames"*, e_1, 0..*)
    - p_{73} = (*"Code"*, e_1, 1..1)

Assim como este metamodelo não entrou no mérito do código dos blocos básicos em linguagem hospedeira, representado por p_{12}, tendo sido o mesmo modelado apenas como uma *string*, o mesmo se aplica a p_{73}, servindo como *slot* para armazenar o código que implementa a função adaptativa em linguagem de programação adaptativa. A restrição r_5 impede funções adaptativas com nome vazio, enquanto r_6 não permite nomes repetidos no mesmo modelo. Já r_7, por sua vez, impede que uma função adaptativa defina um parâmetro com nome vazio, ao mesmo tempo em que r_8 garante que não há dois parâmetros com o mesmo nome.

- r_5: $\forall o \mid$ MCAdaptiveFunction(o) $\rightarrow \neg$ (MCAdaptiveFunction.Name (o) = ε)
- r_6: $\forall o_1, o_2 \mid$ MCAdaptiveFunction(o_1) $\wedge$ MCAdaptiveFunction(o_2) $\wedge$ (MCAdaptiveFunction.Name(o_1) = MCAdaptiveFunction. Name(o_2)) $\rightarrow o_1 = o_2$
- r_7: $\forall a, o \mid$ MCAdaptiveFunction(a) $\wedge$ String(o) $\wedge$ ($o \in$ MCAdaptiveFunction.ParametersNames(a)) $\rightarrow \neg$(o = ε)
- r_8: $\forall a, o_1, o_2 \mid$ MCAdaptiveFunction(a) $\wedge$ String(o_1) $\wedge$ String(o_2) $\wedge$ ($o_1 \in$ MCAdaptiveFunction.ParametersNames(a)) $\wedge$ ($o_2 \in$ MCAdaptiveFunction.ParametersNames(a)) $\wedge$ (String.Value(o_1) = String.Value(o_2))) $\rightarrow o_1 = o_2$

Por definição, os parâmetros das funções adaptativas correspondem a blocos básicos, não podendo ser de outro tipo. Por isso não é necessário prever propriedades para tipos de parâmetros.

A restrição r_9, citada anteriormente, deve garantir que para toda chamada de função adaptativa existente em um modelo, ou seja, instâncias de c_5 (*MCAdaptiveFunctionCall*), sejam informados apenas parâmetros esperados pela função chamada.

- r_9: $\forall a, o, f \mid$ MCAdaptiveFunctionCall(a) $\wedge$ MCAdaptiveFuncParamValue(o) $\wedge$ ($o \in$ MCAdaptiveFunctionCall.ParametersValues(a)) $\wedge$ MCAdaptiveFunction(f) $\wedge$ (MCAdaptiveFunctionCall.AdaptiveFunction(a) = f) $\rightarrow$ MCAdaptiveFuncParamValue.ParameterName(o) $\in$ MCAdaptiveFunction.ParametersNames(f)

Por fim, cria-se a metaclasse c_0, que estabelece os blocos de entrada e saída do programa adaptativo, conforme especificado em [3]. Serve como agregador principal dos blocos, conexões e funções adaptativas.

- c_0 = (*"MCAdaptiveProgram"*, P_0)
  - P_0 = {p_{01}, p_{02}, p_{03}, p_{04}, p_{05}, p_{06}}
    - p_{01} = (*"Name"*, e_1, 1..1)
    - p_{02} = (*"EntryBlock"*, c_1, 1..1)
    - p_{03} = (*"ExitBlock"*, c_1, 1..1)
    - p_{04} = (*"OtherBlocks"*, c_1, 0..*)
    - p_{05} = (*"AdaptiveConnections,* c_2, 1..*)
    - p_{06} = (*"AdaptiveFunctions"*, c_7, 0..*)

Como restrição, um modelo de programa adaptativo construído sobre o metamodelo aqui proposto deve possuir uma e apenas uma instância de c_0. Ou seja, não é permitido que um modelo contenha mais de um programa adaptativo, e nem que haja ausência do mesmo. As restrições r_{10} e r_{11} estabelecem, respectivamente, que existe um programa adaptativo no modelo e que ele é único. Enquanto isso, r_{12} impede que o programa adaptativo tenha nome vazio.

- r_{10}: $\exists o \mid$ MCAdaptiveProgram(o)
- r_{11}: $\forall o_1, o_2 \mid$ MCAdaptiveProgram(o_1) $\wedge$ MCAdaptiveProgram(o_2) $\rightarrow o_1 = o_2$
- r_{12}: $\forall o \mid$ MCAdaptiveProgram (o) $\rightarrow \neg$ (MCAdaptiveProgram.Name (o) = ε)

Um aspecto interessante é que neste metamodelo as conexões que saem dos blocos foram definidas não como propriedades dos próprios blocos (metaclassse c_1), mas sim como elementos externos (metaclasse c_2). Esta forma é coerente com o fato de enxergar a camada de conectores e de adaptatividade da Figura 2 como desacopladas das definições dos blocos, podendo ser remanejadas em cada modelo de programa adaptativo enquanto se reusa blocos já existentes de outros programas prévios. Como restrição, em um modelo de programa adaptativo não pode haver mais de uma instância da metaclasse c_2 que referencia o mesmo bloco na propriedade p_{21}, do contrário pode ocorrer não determinismos em tempo de execução. Para isso introduz-se a restrição r_{13}.

- r_{13}: $\forall o_1, o_2 \mid$ MCAdaptiveConnection(o_1) $\wedge$ MCAdaptiveConnection(o_2) $\wedge$ (MCAdaptiveConnection.From(o_1) = MCAdaptiveConnection.From(o_2)) $\rightarrow o_1 = o_2$

Em resumo, os conjuntos da quíntupla do metamodelo MM_{PA} são:

- C = {c_0, c_1, c_2, c_3, c_4, c_5, c_7}
- Γ = {(c_3,c_4)}
- E = {e_1, e_2}
- R = {r_1, r_2, r_3, r_4, r_5, r_6, r_7, r_8, r_9, r_{10}, r_{11}, r_{12}, r_{13}}

O SBMM propõe uma notação gráfica similar a do MOF e UML, na qual se representa as metaclasses e enumerações por retângulos nomeados. As propriedades são representadas por linhas que ligam a metaclasse ao tipo alvo, que pode ser também uma metaclasse ou enumeração. As linhas são rotuladas pelo nome da propriedade e direcionadas com uma seta ao tipo alvo. Nesta notação, o metamodelo MM_{PA} está representado pela Figura 3.

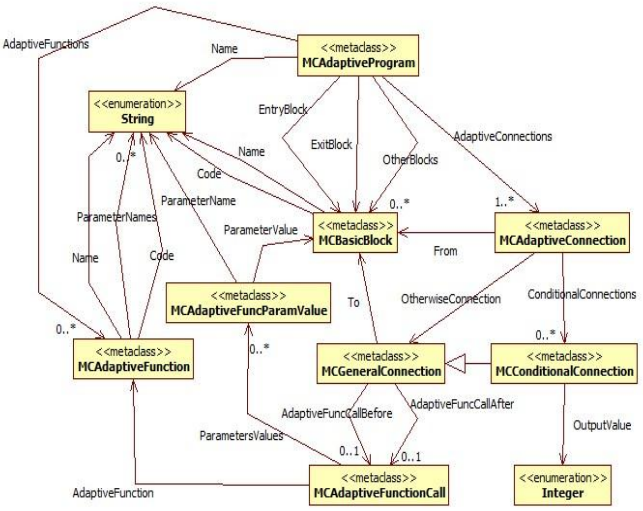

Fig. 3. Notação gráfica do metamodelo.

V. Ferramenta Case para Programas Adaptativos

A ferramenta SBMMTool é um software desenvolvido como parte da pesquisa que originou o SBMM, que permite a edição de metamodelos, modelos e funções de mapeamento, bem como a geração de código-fonte a partir dos modelos baseada nas funções de mapeamento. Trata-se de uma ferramenta meta-CASE. Esse tipo de ferramenta é capaz de gerar ou emular ferramentas CASE customizadas a partir de metamodelos de entrada [7]. Nesse caso, foi inserido na ferramenta o metamodelo para programas adaptativos apresentado na seção IV. Essa inserção é feita através de funcionalidades providas pela ferramenta, tais como inserção e edição de metaclasses, propriedades, enumerações, etc. A Figura 4 ilustra uma tela da ferramenta, na qual são montados os elementos da quíntupla que define o metamodelo definido na seção IV.

As sentenças em lógica de primeira ordem são inseridas na ferramenta com algumas adaptações sintáticas para trabalhar apenas com caracteres ASCII. Por exemplo, o símbolo $\forall$ é denotado por `forall` e $\neg$ é representado por `not`.

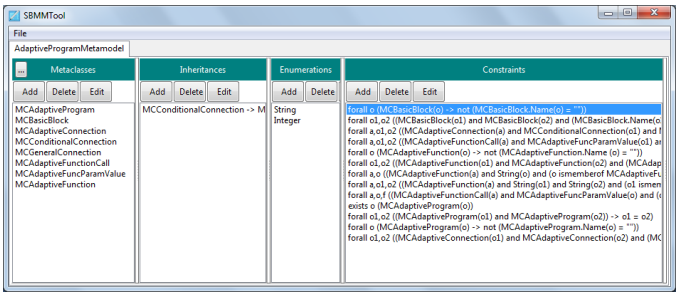

Fig. 4. Tela da ferramenta SBMMTool com o metamodelo AdaptiveProgramMetamodel.

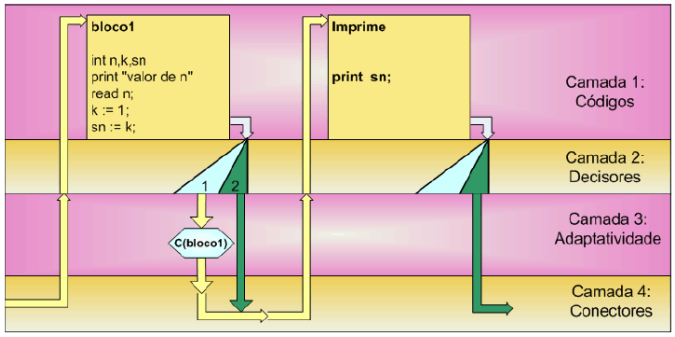

Fig. 5. Modelo de programa adaptativo exemplo [3].

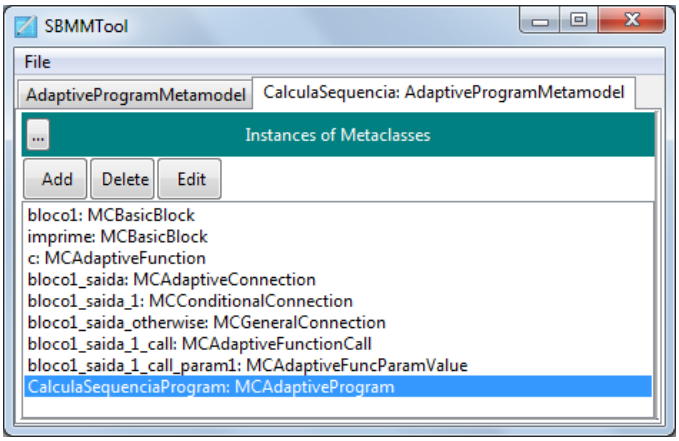

Fig. 6. Modelo de programa adaptativo exemplo editado na ferramenta SBMMTool.

Uma vez que o metamodelo *AdaptiveProgramMetamodel* foi carregado na SBMMTool, é possível criar modelos de programas adaptativos. A ferramenta é responsável por aplicar todas as restrições do conjunto *R* sobre o modelo em edição, garantindo a geração de modelos conformes. A Figura 5 ilustra a tela de edição do modelo de um programa adaptativo exemplo apresentado em [3]. Esse programa calcula o n-ésimo termo da sequência matemática $S(n) = S(n-1) + 2n$ para $n \geq 2$, sendo $S(1) = 1$. A estrutura do programa consiste em dois blocos: *bloco1* e *imprime*. O *bloco1* é responsável por solicitar ao usuário o valor de *n*. Uma função adaptativa *c* é colocada em uma conexão condicional na saída de *bloco1* e insere código no programa para processar mais uma iteração da sequencia quando ainda necessário. O bloco *imprime* é responsável por apresentar o resultado final ao usuário. A Figura 5 mostra o modelo do programa conforme representação gráfica apresentada na seção II [3]. A Figura 6, por sua vez, mostra esse modelo editado na ferramenta SBMMTool.

Como cada bloco básico deve gerar um valor de saída (exceto o último bloco a ser processado), é necessário então atribuir a *bloco1* um valor de saída e um decisor. O decisor direcionará a execução de acordo com o valor de saída. Quando o valor de entrada digitado pelo usuário (variável *n*) é *1*, já se sabe que $S(1) = 1$ e, portanto, o resultado é apenas impresso sem necessidade de cálculo. Para os termos a partir do segundo, a variável *k* de *bloco1* guarda o número de ordem do termo cujo valor foi computado, o qual é armazenado na variável *sn*. A comparação de *k* com *n* permite saber se o termo requerido pelo usuário já foi encontrado, condição que está implementada no decisor. Por convenção, o valor de saída será *1* se o termo *n* ainda não foi atingido e *2* caso contrário. Esse valor é testado pela conexão para determinar o próximo bloco básico a ser processado. O código fonte de *bloco1* e *imprime* pode ser conferido a seguir [3].

```
code bloco1: <
   int n,k,sn;
   print "valor de n";
   read n;
   k := 1;
   sn := k;
   if n>k then bloco1 := 1 else bloco1 := 2;
>;

code imprime: <
   print sn;
>;
```

O bloco *imprime* não necessita de um decisor pois corresponde ao bloco de saída do programa, ou seja, à propriedade *ExitBlock* da única instância de programa adaptativo permitida no modelo , isto é, da metaclasse *MCAdaptiveProgram*.

De acordo com a Figura 5, o valor de saída *1* do bloco *bloco1* indica que o valor informado pelo usuário é maior que *1* e, portanto, é necessário computar os termos seguintes da sequência até o *n*-ésimo. Sendo assim, a conexão que o interliga ao bloco *imprime* chama a função adaptativa *c*, que calculará o *n*-ésimo termo antes de processar o bloco *imprime*.

A montagem da conexão, juntamente com os respectivos valores de decisão e a chamada da função adaptativa para a saída *1*, estão presentes no modelo do programa conforme a Figura 6, mais precisamente de *bloco1_saida* até *bloco1_saida_1_call_param1*. A Figura 7 apresenta uma outra tela da ferramenta com detalhes do preenchimento dos valores das propriedades para a instância do modelo *bloco1_saida*.

A função adaptativa *c*, por sua vez, é responsável por instanciar um novo bloco básico, não presente no modelo original. Também é instanciada uma conexão deste novo bloco básico para *imprime*, passando por um decisor igual ao de *bloco1*, que novamente aciona a função adaptativa *c* para a saída igual a *1*. A condição para a saída deste novo bloco instanciado resultar em *2* é que o valor de *k* atinja o valor de *n*, provocando então a parada do laço de repetição montado através de sucessivas instanciações de blocos básicos pela função adaptativa *c*. O código desta função adaptativa em BADAL pode ser conferido a seguir [3].

```
adaptive function c(i) [generators g] {
   insert [ code g: <
      k := k + 1;
      sn := sn + 2 * k;
      if n>k then g := 1 else g := 2;
      >;
   ];
   insert [ connection from g (case 1: to
imprime perform c(g) before otherwise to
imprime)
   ];
}
```

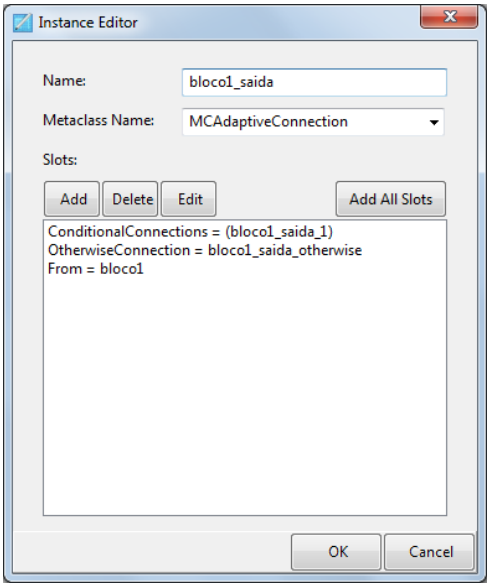

Fig. 7. Editor de instância da ferramenta SBMMTool.

A ferramenta SBMMTool permite a definição e execução de funções de mapeamento do tipo modelo para código, ou seja, funções que tomam modelos como entrada e geram como saída uma cadeia de caracteres representando código-fonte. A definição dessas funções de mapeamento é feita através da linguagem *MOF Model To Text Transformation Language* (MOFM2T) [17]. Esta linguagem é baseada em *templates* de código-fonte dentro dos quais é possível realizar iterações e extração de valores de propriedades dos modelos de entrada.

Embora tenha sido originalmente projetada para trabalhar com metamodelos descritos em MOF, a ferramenta SBMMTool implementa um interpretador baseado em um subconjunto da MOFM2T para operar sobre metamodelos descritos em SBMM. Essa possibilidade existe pois os conceitos principais de metamodelagem e modelagem do SBMM (metaclasse, propriedade, instância e *slot*) são semanticamente os mesmos do MOF, mesmo que com diferenças técnicas nas definições e notações.

Como exemplo, considere-se o diagrama de classes UML da Figura 8, contendo uma única classe.

O *template classToJava* [17], escrito em MOFM2T, é capaz de gerar código Java para a estrutura básica das classes de um modelo, contendo seus atributos e um construtor vazio.

```
[template public classToJava(c : Class)]
class [c.name/] {
   // Attribute declarations
   [for(a : Attribute | c.attribute)]
   [a.type.name/] [a.name/];
   [/for]

   // Constructor
   [c.name/]() {}
}
[/template]
```

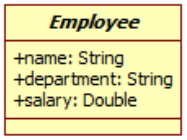

Fig. 8. Exemplo de diagrama de classes.

As expressões entre colchetes fazem com que o interpretador realize algum processamento, tais como iterações ou extração de valores de propriedades do modelo, para então gerar a saída de caracteres naquela posição. O exemplo acima contém um laço do tipo *for*, que itera sobre os atributos da classe, e também extrações de valores de propriedades de tipos primitivos. A saída gerada por esta função de mapeamento, quando aplicada sobre o modelo da Figura 8, está reproduzida a seguir.

```
class Employee {
  // Attribute declarations
  String name;
  String department;
  Double salary;

  // Constructor
  Employee() {}
}
```

Uma função de mapeamento em MOFM2T capaz de gerar código-fonte BADAL para modelos baseados no metamodelo apresentado na seção IV foi criada como parte deste trabalho. Por questões de espaço, apenas o trecho do código desta função que gera o cabeçalho do programa e o fechamento de sua estrutura principal está reproduzido a seguir. O código completo está disponível para *download* no link http://www.p2s.com.br/team/scanovas/wta2015/amtobadal.txt.

```
[template public Main()]
  [if(MCAdaptiveProgram[0])]
  [file('AdaptiveProgram.txt',0)]ADAPTIVE MAIN
\[NAME = [echo(MCAdaptiveProgram[0].Name) /],
ENTRY = [value(MCAdaptiveProgram[0].
EntryBlock.Name) /], EXIT = [echo(
MCAdaptiveProgram[0].ExitBlock.Name) /] ] IS

  ...

  END MAIN.
  [/file]
  [/if]
[/template]
```

Com isso, a SBMMTool passa a se comportar como uma ferramenta CASE para elaboração de modelos de programas adaptativos, com possibilidade de geração automática de código BADAL. O modelo do programa de cálculo de sequência apresentado nesta seção pode ser convertido em código-fonte através da execução da transformação baseada

nesta função de mapeamento, podendo o resultado ser obtido pelo link http://www.p2s.com.br/team/scanovas/wta2015 /calcseq.txt.

VI. Resultados e Discussão

O presente trabalho apresenta a utilização da ferramenta SBMMTool para edição de modelos de programas adaptativos. Esse resultado é obtido graças às equações e sentenças apresentadas na seção IV, que especificam formalmente um metamodelo para programas adaptativos, adicionando restrições sobre modelos com relação ao trabalho apresentado em [15].

Existem diversos tipos de ferramentas CASE, desde aquelas que servem apenas para desenhar modelos baseados em notações gráficas, tais como UML, até ferramentas capazes de transformar modelos em outros modelos ou em código-fonte, ponto chave para aplicação da MDE. A ferramenta SBMMTool provê a capacidade deste tipo de transformação através de um interpretador de um subconjunto da linguagem MOFM2T [17], permitindo a geração automática de código a partir de modelos.

No entanto, um modelo descrito no metamodelo proposto será capaz de gerar código adaptativo apenas no que diz respeito à estrutura dos blocos e conexões. A implementação dos blocos e funções adaptativas em si foram modeladas como propriedades *String*, ou seja, a SBMMTool ou outra ferramenta que permita o programador editar um modelo conforme ao metamodelo apresentado na seção IV considera que essas implementações são texto livre, devendo o programador preencher código na linguagem hospedeira e linguagem adaptativa de interesse. Futuras extensões do metamodelo proposto podem considerar detalhes dos aspectos de implementação ao menos das funções adaptativas, permitindo que o modelo contenha informações completas sobre sua implementação nativamente, sem depender de nenhuma linguagem específica baseada em texto tal como a própria BADAL.

Outro aspecto a ser considerado é a forma com a qual os modelos são editados na ferramenta. Por ser genérica e trabalhar com qualquer metamodelo válido, de programa adaptativo ou não, a ferramenta SBMMTool permite a edição de modelos através da inserção de instâncias de metaclasses e preenchimento de propriedades em listas. Essa forma não é muito conveniente, sendo as notações gráficas mais adequadas para facilitar o trabalho da construção e entendimento de modelos. A própria UML não teria obtido sucesso se não fosse primariamente baseada em uma sintaxe concreta composta por desenhos. Trabalha-se também em um editor de sintaxe concreta para a SBMMTool, que permite editar os símbolos gráficos utilizados para cada metaclasse. Desse modo, as instâncias inseridas atualmente em forma de listas poderiam ser arrastadas para um editor gráfico que permitiram a modelagem do programa adaptativo na representação gráfica apresentada na seção II.

VII. Conclusão

O presente trabalho mostra um caminho em direção à utilização da abordagem MDE para o desenvolvimento de programas adaptativos. A MDE é considerada por alguns a grande tendência futura da engenharia de software [16], embora atualmente pouco explorada na prática. Programas adaptativos representam uma nova abordagem para resolver problemas. O lado prático do caminho apresentado consiste na utilização de ferramentas CASE para programas adaptativos com possibilidade de gerar código, nas quais os modelos servem não só como documentação, mas também como artefatos básicos de construção dos programas. Tal objetivo foi colocado inicialmente por [15]. Como um passo futuro, funções de mapeamento adicionais podem ser criadas para gerar o programa para distintas plataformas ou linguagens alvo a partir do mesmo modelo, não somente em BADAL

VIII. Referências

[1] OMG, MDA Guide Version 1.0.1, 2003. Disponível em: http://www.omg.org/cgi-bin/doc?omg/03-06-01.

[2] Ambler, S., The Object Primer: Agile Modeling-Driven Development with UML 2.0. New York: Cambridge University Press, 2004.

[3] Silva, S. R. B., Software Adaptativo: Método de Projeto, Representação Gráfica e Implementação de Linguagem de Programação. Dissertação (Mestrado). Escola Politécnica da Universidade de São Paulo, 2010.

[4] Neto, J. J., *Adaptive Rule-Driven Devices - General Formulation and Case Study*. Lecture Notes in Computer Science. Watson, B.W. and Wood, D. (Eds.): Implementation and Application of Automata 6th International Conference, CIAA 2001, Vol. 2494, Pretoria, South Africa, July 23-25, Springer-Verlag, 2001, pp. 234-250.

[5] Hirakawa, A. R., Saraiva, A. M., Cugnasca, C. E., *Adaptive Automata Applied on Automation and Robotics (A4R)*. Latin America Transactions, IEEE, 2007. 5(7), 539-543.

[6] Pelegrini, E. J., Códigos Adaptativos e Linguagem para Programação Adaptativa: Conceitos e Tecnologia. Dissertação (Mestrado). Escola Politécnica da Universidade de São Paulo, 2009.

[7] De Lara, J., Vangheluwe, H., Using AToM as a Meta-CASE Tool. In: Proceedings of the 4th International Conference on Enterprise Information Systems (ICEIS), 2002.

[8] Rutle et al., A formal approach to the specification and transformation of constraints in MDE. The Journal of Logic and Algebraic Programming 81, no. 4, 2012.

[9] Gessenharter, D.; Rauscher, M., Code generation for UML Activity Diagrams. In Modeling Foundations and Applications (pp 205-220), Springer, 2011.

[10] Marchetti, G., Um método de transformação de modelos UML para a inclusão de componentes de frameworks com o uso de planejador lógico. Dissertação (Mestrado), Escola Politécnica da USP, 2012.

[11] Pastor, O.; Molina, J., Model-driven architecture in practice. New York: Springer-Verlag, 2010.

[12] Jouault, F.; Bézivin, J., KM3: A DSL for metamodel specification. In Formal Methods for Open Object-Based Distributed Systems (pp. 205-220), Springer, 2006.

[13] Jackson et al., Automatically reasoning about metamodeling. Software & Systems Modeling, 2013.

[14] Alanen, M.; Porres, I., A metamodeling language supporting subset and union properties. Software & Systems Modeling 7, 2008.

[15] Canovas, S.; Cugnasca, C., Um Metamodelo para Programas Adaptativos Utilizando SBMM. In: Oitavo Workshop de Tecnologias Adaptativas, Escola Politécnica da Universidade de São Paulo, São Paulo, 2014.

[16] Yang, D.; Liu, M.; Wang, S., Object Oriented Methodology Meets MDA Software Paradigm. In: Software Engineering and Service Science (ICSESS), IEEE 3rd International Conference on (pp. 208-211), 2012.

[17] OMG, MOF Model To Text Transformation Language v1.0, 2008. Disponível em: http://www.omg.org/spec/MOFM2T/1.0.
[18] Contier, A.; Padovani, D.; Neto, J. J., Linguístico: Usando Tecnologia Adaptativa para a Construção de Um Reconhecedor Gramatical. In: Oitavo Workshop de Tecnologias Adaptativas, Escola Politécnica da Universidade de São Paulo, São Paulo, 2014.
[19] Silva, A. P.; Silva Filho, R. I.; Rodrigues, F. A.. Swarm Robotics: comportamento Adaptativo Aplicado ao problema de dispersão de pássaros em áreas agrícolas. In: Oitavo Workshop de Tecnologias Adaptativas, Escola Politécnica da Universidade de São Paulo, São Paulo, 2014.

Sergio R. M. Canovas nasceu em São Paulo, Brasil, em 1981. Graduou-se e recebeu o título de mestre em Engenharia Elétrica pela Universidade de São Paulo (USP), São Paulo, Brasil em 2003 e 2006, respectivamente. Em 2011 ingressou no programa de doutoramento em Engenharia Elétrica pela mesma universidade, sendo pesquisador no Laboratório de Automação Agrícola (LAA). Seus interesses de pesquisa incluem redes de controle, sistemas ERP, MDE/MDA, formalismos para metamodelagem e transformação de modelos de software.

Carlos E. Cugnasca graduou-se em Engenharia Elétrica na Escola Politécnica da Universidade de São Paulo (EPUSP), Brasil, em 1980. Na mesma instituição, obteve o seu mestrado (1988), doutorado (1993) e Livre-Docência (2002). Suas principais atividades de pesquisas incluem instrumentação inteligente, automação agrícola e agricultura de precisão. É professor do Departamento de Engenharia de Computação e Sistemas Digitais da EPUSP desde 1988, ocupando atualmente a função de Professor Associado 3. Vem desenvolvendo pesquisas sobre redes de controle, redes de sensores sem fio, eletrônica embarcada e aplicações de computação pervasiva.